RSRE
MEMORANDUM No. 4225

ROYAL SIGNALS & RADAR ESTABLISHMENT

MATHEMATICAL EQUIVALENCE IN A PRIMITIVE ELLA

Author: M B Davies

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

RSRE MEMORANDUM No. 4225

UNLIMITED

Royal Signals and Radar Establishment

Memorandum 4225

Title: Mathematical Equivalence in a Primitive Ella

Author: M B Davies (student scientist at RSRE 1987-1988)

Date: August 1988

Summary

The mathematical language $\mathcal{L}$ is defined, which represents circuits in a primitive subset of Ella containing delays, pairing, CASE (*multiplexor*) expressions, and recursive or feedback expressions. It is shown how by reduction of expressions in this language to an approximated finite form, equivalence between expressions can be tested in finite time.

Contents

1. Introduction

The language $\mathcal{L}$ has been defined so that all Ella programs can be expressed in it. Although it is very primitive and does not even have such concepts as functions or indexing, mechanisms already exist for the transformation of general, high-level Ella into a form not far removed from the $\mathcal{L}$ expression (see [1] and [2]). However, once reduced into $\mathcal{L}$, Ella programs can not be recovered in their original form. For the purposes of equivalence testing there is no need for this.

Expressions in $\mathcal{L}$ are defined inductively from the basic units of type constants and variables (input signals). Since the definition is inductive, all functions which I create to operate on such expressions are also defined using structural induction. However, such functions operate from the top down, breaking expressions into their component parts as they are applied; while the expressions themselves are defined from the roots up. This distinction is in practice obvious and in only one case causes problems.

The mathematical notation used in the definitions is mostly standard; Greek letters are used to represent general expressions; the letter t usually represents a general time (natural number); k is used for a general type constant; v and w for general variable names; s for a general type. Capital letters usually indicate sets or sequences. Sequences are ordered sets, and are listed using angle bracket delimiters, eg

$$S = \langle S_1 ... S_n \rangle$$

Sequences are indexed using a subscript notation, eg A_i is the i^{th} element of the sequence A.

A notation is needed to represent the set of pairs of items. For example, a compound type is either a basic type or a pair of other compound types. I denote this sort of structure $\times^{\mathbf{S}}$. Such sets are formally defined in Appendix **A**.

2. Expressions in $\mathcal{L}$

$\mathcal{L}_{\mathbf{S}}$ is a structure consisting of:

$\mathbf{S}$ types, eg $\mathbf{S} = \{\mathsf{BOOL}, \mathsf{NUM}, \mathsf{DATA}\}$.
$\{S_s\}_{s\in \mathbf{S}}$ constants, eg $S_{\mathsf{BOOL}} = \langle \mathsf{T}, \mathsf{F}\rangle$. Sequences S_s disjoint.
$\{V_s\}_{s\in \mathbf{S}}$ variables, eg a,b,c,in,cntrl-line $\in V_{\mathsf{BOOL}}$.
Sequences V_s disjoint.
$\mathbf{E}$ expressions
$\{\bot_s\}_{s\in \mathbf{S}}$ bottom

Each language $\mathcal{L}_{\mathbf{S}}$ contains all the types (with their corresponding enumerated type constants) and variable names that can be used in expressions in that language. Expressions in different languages $\mathcal{L}_{\mathbf{S}}$ and $\mathcal{L}_{\mathbf{S}'}$ are not directly comparable; they must be re-expressed in the superset language $\mathcal{L}_{\mathbf{S}\cup\mathbf{S}'}$.

For each type s in the language, there is a 'bottom', $\bot_s$. This is the undefined value and is not accessible to the user. Its only function is to serve as the limit value in the approximation of recursive loops, as will be seen later.

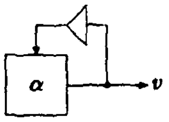
Variables are ordered, perhaps alphabetically. This is to help establish a unique form for expressions, as is shown at the end of this paper.

2.1 DEFINITION $\mathbf{E}$ is the least set s.t.

$$\begin{array}{rll}
 & S_s \subset \mathbf{E} & \forall s \in \mathbf{S} \\
 & V_s \subset \mathbf{E} & \forall s \in \mathbf{S} \\
\text{delay} & \Delta_t\alpha \in \mathbf{E} & \forall \alpha \in \mathbf{E},\ \forall t \in S_s \text{ s.t. type}(\alpha) = s \\
\text{pair} & (\alpha, \beta) \in \mathbf{E} & \forall \alpha, \beta \in \mathbf{E} \\
\text{case} & \Box\alpha{:}\mathcal{A} \in \mathbf{E} & \forall \alpha \in \mathbf{E},\ \forall \mathcal{A} \subset \mathbf{E} \text{ s.t. welldef}_{\Box}(\alpha, \mathcal{A}) \\
\text{recurse} & \mu v.\alpha \in \mathbf{E} & \forall v \in V_s,\ \forall \alpha \in \mathbf{E} \text{ s.t. welldef}_{\mu}(v, \alpha)
\end{array}$$

The above defines the syntax of $\mathcal{L}_{\mathbf{S}}$ expressions. An expression in $\mathbf{E}$ is either a constant or variable, or constructed out of such by the pair, delay, case or recursion operators. Illegal expressions, such as badly-typed case expressions, are not allowed: see Appendix **A** for definitions of 'type', 'welldef$_{\Box}$' and 'welldef$_{\mu}$'.

Pairing is the only form of structuring provided: tuples are not needed. Indexing of pairs is also not required since obviously in an expressional language without functions, $(\alpha, \beta)[1] = \alpha$ and $(\alpha, \beta)[2] = \beta$. A case statement $\Box\alpha{:}\mathcal{A}$ outputs at any time unit, one of the expressions in the sequence $\mathcal{A}$. The expression chosen is indexed by the value of the chooser α at that time. The recursive operator $\mu v.\alpha$ outputs the value of v, where v is let equal to α. Of course, α usually contains some delayed instance of the dummy variable v, so that a feedback loop is produced. Pictorially, this can be represented as:



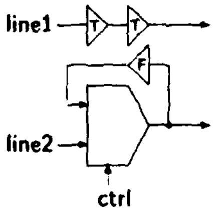
An example of a legal **E** expression:

$$(\Delta_T\Delta_T\mathrm{line1}, \mu\mathrm{register}.\Box\mathrm{ctrl}:(\Delta_F\mathrm{register}, \mathrm{line2}))$$

This would correspond to the Ella *unit* (see [3]):

```
BEGIN
     FN REG = (bool: register) -> bool:
          CASE ctrl OF t: DEL{f} register,
                       f: line2
          ESAC.
     MAKE REG: reg.
     LET register = reg.
     JOIN register -> reg.
     OUTPUT (DEL{t} DEL{t} line1, register)
END
```

Or in graphical form:



2.2 DEFINITION $\text{evaluate}_{t,V} : \mathbf{E} \longrightarrow \times \bigcup_{s \in \mathbf{S}} s$

$t \in \mathbf{N}$, V *valuation*

Returns the value of an expression given a valuation V.

$$\begin{array}{lll}
\text{evaluate}_{t,V}(k) & \mathrel{\hat=} & k, \quad \forall k \in S_s \\
\text{evaluate}_{t,V}(v) & \mathrel{\hat=} & \textit{the value of } v \textit{ in valuation } V \textit{ at time } t. \\
\text{evaluate}_{t,V}(\Delta_k \alpha) & \mathrel{\hat=} & \text{evaluate}_{t-1,V}(\alpha) \quad t > 0 \\
 & & k \qquad\qquad\qquad\quad\;\; t = 0 \\
\text{evaluate}_{t,V}((\alpha, \beta)) & \mathrel{\hat=} & (\text{evaluate}_{t,V}(\alpha), \text{evaluate}_{t,V}(\beta)) \\
\text{evaluate}_{t,V}(\Box\alpha{:}\mathcal{A}) & \mathrel{\hat=} & \text{evaluate}_{t,V}(\mathcal{A}_n) \\
\quad \text{where } n & \mathrel{\hat=} & i \quad \text{s.t. } (\overline{S}_{\text{type}(\alpha)})_i = \text{evaluate}_{t,V}(\alpha) \\
\text{evaluate}_{t,V}(\mu v.\alpha) & \mathrel{\hat=} & \text{evaluate}_{t,V'}(\alpha) \\
\quad \text{where } V' & \mathrel{\hat=} & V \cup \bigcup_{n=1}^{t-1} \{(v_{t-n}, \text{evaluate}_{t-n,V}(\mu v.\alpha))\}
\end{array}$$

This function defines the result of evaluating an expression; in other words, it gives the semantics of **E**. A *valuation* is simply a set which contains a constant assignment to each external variable in the expression, at every time unit: that is, pairs of the form (v_t, k) where variable v is to have value k at time t.

The delay operator $\Delta_k\alpha$ delays signal α by one time unit. At time 0 it outputs instead its initialization constant k.

As an example of the evaluation of the case expression, consider $\Box(\mathsf{a}, \mathsf{b}){:}\mathcal{A}$ where a and b are Boolean variables. The equivalent Ella for this expression is:

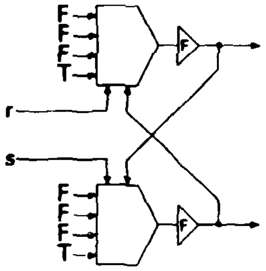
```
CASE (a,b) OF (t,t):  $\mathcal{A}_1$,
              (f,t):  $\mathcal{A}_2$,
              (t,f):  $\mathcal{A}_3$,
              (f,f):  $\mathcal{A}_4$
ESAC
```

For each possible value of the chooser (a, b), there must be a corresponding expression in $\mathcal{A}$. That is, there is no partial choice in $\mathcal{L}$. The order in which expressions in $\mathcal{A}$ are chosen depends on the orderering on the basic enumerated types. The above example assumes that $\mathsf{BOOL} = \langle \mathsf{T}, \mathsf{F} \rangle$, so the 'True's come first. $\overline{S}_{(\mathsf{BOOL},\mathsf{BOOL})}$ is the sequence which contains all possible constant values of (a, b); it is defined in Appendix **A** to equal $\langle (\mathsf{T}, \mathsf{T}), (\mathsf{F}, \mathsf{T}), (\mathsf{T}, \mathsf{F}), (\mathsf{F}, \mathsf{T}) \rangle$. As you can see, it is simply the cross product of S_{BOOL} with itself, with the convention that the first position changes more rapidly.

A recursive expression is evaluated by adding its previous values to the valuation of its dummy variable.

It is possible, though perhaps lengthy, to express any Ella circuit in **E**, no matter how convoluted it is. For example, here is an RS latch:

$$(\mu v.\Delta_F\Box(r,\ \mu w.\Delta_F\Box(v, s):\langle F, F, F, T\rangle):\langle F, F, F, T\rangle,$$
$$\mu w.\Delta_F\Box(\mu v.\Delta_F\Box(r, v):\langle F, F, F, T\rangle,\ s):\langle F, F, F, T\rangle)$$



2.3 DEFINITION ≡

Equivalence in **E**.

$$\alpha \equiv \beta \text{ iff } \text{evaluate}_{t,V}(\alpha) = \text{evaluate}_{t,V}(\beta)$$
$$\forall t = 0 \ldots \infty$$
$$\forall \text{ valuations } V$$

Clearly two expressions are equivalent, no matter how they are expressed, if at every time they evaluate to the same result for every combination of input values.

3. Reduction into Finite Expressions $\mathbf{E}^\star$

Equivalence testing in $\mathbf{E}$ is not feasible since the behaviour of compared circuits must be checked at all time steps, from 0 to ∞. Is there a way to determine circuit equivalence in finite time? Since the definition of $\mathbf{E}$ is by structural induction, we know that all expressions in $\mathbf{E}$ have finite 'size'. Furthermore, we can assume that all types in $\mathcal{L}_{\mathbf{S}}$ are also finite, since they are meant to represent enumerated Ella types. So it is clear that no circuit expressible in $\mathbf{E}$ can have a behaviour that requires infinite time to characterize. That is, we should be able to find a bound on the performance of a circuit, after which time we have observed all possible behaviour from that circuit. The following two functions find this bound.

3.4 DEFINITION $\quad \mathrm{depth} : \mathbf{E} \longrightarrow \mathbf{N}$

Minimum time until all external signals are reaching expression.

$$\begin{aligned}
\mathrm{depth}(k) &\mathrel{\widehat{=}} 0, \quad \forall k \in S_s \\
\mathrm{depth}(v) &\mathrel{\widehat{=}} 0, \quad \forall v \in V_s \\
\mathrm{depth}(\Delta_k \alpha) &\mathrel{\widehat{=}} 1 + \mathrm{depth}(\alpha) \\
\mathrm{depth}((\alpha, \beta)) &\mathrel{\widehat{=}} \max\{\mathrm{depth}(\alpha), \mathrm{depth}(\beta)\} \\
\mathrm{depth}(\Box\alpha{:}\mathcal{A}) &\mathrel{\widehat{=}} \max\left\{\mathrm{depth}(\alpha), \max \bigcup_{i=1}^{\mathrm{size}(\mathrm{type}(\alpha))}\{\mathrm{depth}(\mathcal{A}_i)\}\right\} \\
\mathrm{depth}(\mu v.\alpha) &\mathrel{\widehat{=}} \mathrm{depth}(\alpha)
\end{aligned}$$

After observing a circuit for 'depth' time units, a state is reached where no more delay initialization constants are affecting the output. The definition is quite simple, and merely counts the maximum number of delays to any expression root.

3.5 DEFINITION $\quad \mathrm{period}_{V,d} : \mathbf{E} \longrightarrow \mathbf{N}$

$V \subset \bigcup_{s \in \mathbf{S}} V_s,\ d \in \mathbf{N}$

Period of longest sequence that expression can produce with constant input.

$$\begin{aligned}
\mathrm{period}_{V,d}(k) &\mathrel{\widehat{=}} 1, \quad \forall k \in S_s \\
\mathrm{period}_{V,d}(w) &\mathrel{\widehat{=}} 1, \quad \forall w \notin V \\
\mathrm{period}_{V,d}(v) &\mathrel{\widehat{=}} (\mathrm{size}(\mathrm{type}(v)))^d, \quad \forall v \in V \\
\mathrm{period}_{V,d}(\Delta_k \alpha) &\mathrel{\widehat{=}} \mathrm{period}_{V,d+1}(\alpha) \\
\mathrm{period}_{V,d}((\alpha, \beta)) &\mathrel{\widehat{=}} \mathrm{period}_{V,d}(\alpha) \times \mathrm{period}_{V,d}(\beta) \\
\mathrm{period}_{V,d}(\Box\alpha{:}\mathcal{A}) &\mathrel{\widehat{=}} \mathrm{period}_{V,d}(\alpha) \times \max\{\textstyle\prod_{i=1}^{\#P} P_i\} \\
&\qquad \forall P \subseteq \{\mathrm{period}_{V,d}(\mathcal{A}_1), \ldots, \mathrm{period}_{V,d}(\mathcal{A}_{\mathrm{size}(\mathrm{type}(\alpha))})\} \\
&\qquad \text{s.t. } \#P \leq \mathrm{period}_{V,d}(\alpha) \\
\mathrm{period}_{V,d}(\mu w.\alpha) &\mathrel{\widehat{=}} \mathrm{period}_{V \cup \{w\},d}(\alpha)
\end{aligned}$$

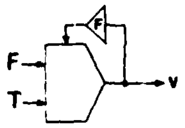
The behaviour of any circuit, as explained above, must be in some sense periodic: after a long enough time, the output will begin to repeat, assuming external signals are held constant. This latter can be assumed because, in equivalence checking, external signals are given a constant valuation anyway. After time 'depth', if all external signals are held constant, only the recursive operator is capable of originating a non-constant output. The maximum possible period of a recursive loop depends upon the number of delays in its feedback loop, and the size of its type. For example, a Boolean loop with a single feedback delay has a maximum possible period of 2, corresponding to the waveform T,F,T,F, ... In general, a single-delay loop has maximum period equal to the size of its enumerated type. A Boolean loop with two feedback delays can at most output a sequence of period 4, eg T,T,F,F,T,T,F,F, ... Hence line 3 in the above definition: the maximum period of a d-delayed loop is the size of the type raised to the power d.

The maximum period of a pair depends on the periods of its components. For example, consider the pair of booleans (a, b). Suppose a produces the waveform T,F, ... and b T,T,F,F, ... Then the pair has period 4: (T, T), (F, T), (T, F), (F, F), ... However, if b had instead waveform T,T,F, ..., ie period 3, the pair would have period 6: (T, T), (F, T), (T, F), (F, T), (T, T), (F, F), ... Obviously, the period of the pair is in general the lowest common multiple of the periods of its components. Unfortunately, the multiple of the periods of the components must actually be taken, since the maximum possible periods only are known. For example, if the maximum periods of the components are 2 and 4 as above, then the lowest common multiple is 4. However, the actual periods might be 2 and 3, ie the second component might not be producing as large a sequence as it could in theory. The combined period would then be 6. To ensure a large enough estimate, 2 times 4 = 8 must be chosen.

The period of the case expression $\square\alpha:\mathcal{A}$ is more difficult. Basically, if the period of the chooser is p, then the output of the case can only cycle between at most p expressions in $\mathcal{A}$. If p is larger than the size of $\mathcal{A}$, then obviously all expressions in $\mathcal{A}$ can affect the output, so the total period is the multiple of all of these, including p itself. If p is smaller than the size of $\mathcal{A}$, the total period is the multiple of p expressions chosen from $\mathcal{A}$ together with p. We have no way of knowing which p expressions this will be, so we have to choose the set which yields the maximum multiple.

The period of a recursive loop has been explained above. The 'period' function keeps a track of all dummy variables it has encountered on its way into an expression, so that it can deal with loops within loops. The 'period' function is initially called with the set V empty, and $d = 0$.

Here is an example of the period of a simple Boolean loop:



$$\begin{aligned}\mathrm{period}_{\emptyset,0}(\mu \mathrm{v}.\Box\Delta_{\mathrm{F}}\mathrm{v}{:}\langle \mathrm{F}\,\mathrm{T}\rangle) &= \mathrm{period}_{\{\mathrm{v}\},0}(\Box\Delta_{\mathrm{F}}\mathrm{v}{:}\langle \mathrm{F}\,\mathrm{T}\rangle)\\ &= \mathrm{l.c.m.}\left\{\mathrm{period}_{\{\mathrm{v}\},0}(\Delta_{\mathrm{F}}\mathrm{v}),\ \mathrm{period}_{\{\mathrm{v}\},0}(\mathrm{F}),\ \mathrm{period}_{\{\mathrm{v}\},0}(\mathrm{T})\right\}\\ &= \mathrm{l.c.m.}\left\{\mathrm{period}_{\{\mathrm{v}\},1}(\mathrm{v}),\ 1,\ 1\right\}\\ &= 2^1 = 2\end{aligned}$$

Once the period and depth of an expression have been found, the sum of these two values is the time needed to observe all possible behaviour from the expression. So equivalence between two expressions can been determined by evaluating them for all possible input combinations for times 0 up to this bound. Although this technique can be implemented in finite time, for circuits of any complexity it would still take far too long in practice, since every variable must be given a constant valuation for these times. We would like to be able to prove equivalence algebraically.

One way of attempting this is to reduce expressions in **E** into another form without recursive operators. This is done by making use of the identity:

$$\mu v.\alpha = \alpha[v\backslash \mu v.\alpha]$$

This is an expansion of the loop, in which every occurrence of v within the loop is substituted by the loop itself. In some senses the loop $\mu v.\alpha$ is actually the limit of the series of expansions:

$$\begin{array}{l}\perp_s\\ \alpha[v\backslash\perp_s]\\ \alpha[v\backslash\alpha[v\backslash\perp_s]]\\ \alpha[v\backslash\alpha[v\backslash\alpha[v\backslash\perp_s]]]\\ \ \vdots\end{array}$$

If the period of the loop $\mu v.\alpha$ is p, the $(p+1)^{\mathrm{th}}$ line of the above series represents a combinatorial circuit that behaves as the loop does, up until time p, whenupon it produces bottoms. It is therefore a finite approximation to the loop. If we expand all loops within an expression in this way, to the maximum period + depth of the whole expression, then we have formed a new, combinatorial expression that will do everything that the original expression did, but not repeatedly. This is enough to decide equivalence or otherwise between expressions.

The expanded expressions are in a set which I shall call $\mathbf{E}^\star$. $\mathbf{E}^\star$ is like $\mathbf{E}$ except that there are no longer any recursive operators and bottoms may appear. Notice that we need bottoms to show that an expression *is* an approximation, and was not originally a combinatorial circuit anyway. The following functions provide transformation from $\mathbf{E}$ to $\mathbf{E}^\star$. A formal definition of $\mathbf{E}^\star$ appears in Appendix **A**.

3.6 DEFINITION $\quad \mathrm{reduce}_d : \mathbf{E} \longrightarrow \mathbf{E}^\star$

$d \in \mathbf{N}$

Approximates $\mathbf{E}$ *expressions to depth* d *in* $\mathbf{E}^\star$.

$$\begin{aligned}
&\mathrm{reduce}_d(k) \;\hat{=}\; k, \quad \forall k \in S_s \\
&\mathrm{reduce}_d(v) \;\hat{=}\; v, \quad \forall v \in V_s \\
&\mathrm{reduce}_d(\boldsymbol{\Delta}_k\alpha) \;\hat{=}\; \boldsymbol{\Delta}_k\mathrm{reduce}_{d-1}(\alpha) \\
&\mathrm{reduce}_d((\alpha,\beta)) \;\hat{=}\; (\mathrm{reduce}_d(\alpha), \mathrm{reduce}_d(\beta)) \\
&\mathrm{reduce}_d(\Box\alpha{:}\mathcal{A}) \;\hat{=}\; \Box\mathrm{reduce}_d(\alpha){:}\, \langle \mathrm{reduce}_d(\mathcal{A}_i)\rangle_{i=1}^{\mathrm{size}(\mathrm{type}(\alpha))} \\
&\mathrm{reduce}_d(\mu v.\alpha) \;\hat{=}\; \mathrm{expand}_{d,v,\alpha}(\alpha)
\end{aligned}$$

'*Reduce*' is actually a trivial function: all it does it find recursive subexpressions to apply 'expand' (below) to.

3.7 DEFINITION $\quad \mathrm{expand}_{d,v,\gamma} : \mathbf{E} \longrightarrow \mathbf{E}^\star$

$d \in \mathbf{N},\ v \in V_s,\ \gamma \in \mathbf{E}$

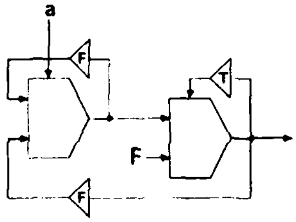
Approximates a loop by expanding d *times.*

$$\begin{aligned}
&\mathrm{expand}_{d,v,\gamma}(k) \;\hat{=}\; k, \quad \forall k \in S_s \\
&\mathrm{expand}_{d,v,\gamma}(w) \;\hat{=}\; w, \quad \forall w \in V_s,\ w \neq v \\
&\mathrm{expand}_{d,v,\gamma}(v) \;\hat{=}\; \bot_{\mathrm{type}(v)}, \quad d < 0 \\
&\mathrm{expand}_{d,v,\gamma}(v) \;\hat{=}\; \mathrm{expand}_{d,v,\gamma}(\gamma), \quad d \geq 0 \\
&\mathrm{expand}_{d,v,\gamma}(\boldsymbol{\Delta}_k\alpha) \;\hat{=}\; \boldsymbol{\Delta}_k\mathrm{expand}_{d-1,v,\gamma} \\
&\mathrm{expand}_{d,v,\gamma}((\alpha,\beta)) \;\hat{=}\; (\mathrm{expand}_{d,v,\gamma}(\alpha), \mathrm{expand}_{d,v,\gamma}(\beta)) \\
&\mathrm{expand}_{d,v,\gamma}(\Box\alpha{:}\mathcal{A}) \;\hat{=}\; \Box\mathrm{expand}_{d,v,\gamma}(\alpha){:}\, \left\langle \mathrm{expand}_{d,v,\gamma}(\mathcal{A}_i)\right\rangle_{i=1}^{\mathrm{size}(\mathrm{type}(\alpha))} \\
&\mathrm{expand}_{d,v,\gamma}(\mu w.\alpha) \;\hat{=}\; \mathrm{expand}_{d,v,\gamma}(\mathrm{expand}_{d,w,\alpha}(\alpha))
\end{aligned}$$

'Expand' keeps a record of the loop variable it is currently expanding, the number of times it has still got to expand it, and the original expression which is used to replace the loop variable. When it encounters inner loops, the function first expands them to the required depth before continuing with the expansion of the outer loop.

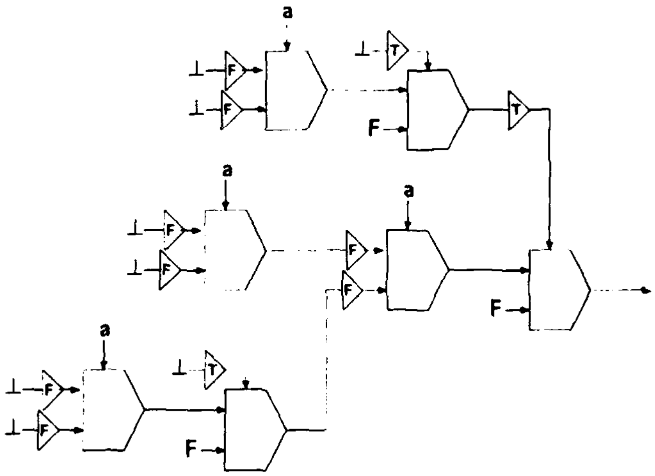
Here is an example of the expansion of a double-loop expression:

$$\mu v.\Box\Delta_T v\colon \langle \mu w.\Box a\colon\langle \Delta_F w, \Delta_F v\rangle,\ F\rangle$$



Expanded once, this becomes:

$$\begin{aligned}&\Box\Delta_T\Box\Delta_T\bot_{BOOL}\colon\langle\Box a\colon\langle\Delta_F\bot_{BOOL},\Delta_F\bot_{BOOL}\rangle,\ F\rangle\colon\\ &\quad\langle\Box a\colon\langle\Delta_F\Box a\colon\langle\Delta_F\bot_{BOOL},\Delta_F\bot_{BOOL}\rangle,\\ &\qquad\Delta_F\Box\Delta_T\bot_{BOOL}\colon\langle\Box a\colon\langle\Delta_F\bot_{BOOL},\Delta_F\bot_{BOOL}\rangle,\ F\rangle\rangle,\ F\rangle\end{aligned}$$



4. Equivalence in $\mathbf{E}^\star$

In this section I define a series of functions which reduce expressions in $\mathbf{E}^\star$ to a canonical form. Once in this form, expressions are unique, ie equivalent expressions are equal.

4.8 DEFINITION pushdown : $\mathbf{E}^\star \longrightarrow \mathbf{E}^\star$

Pushes delays down into an expression.

$\forall n \geq 0$

$$\begin{aligned}
&\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}k) \mathrel{\widehat{=}} \mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}k, \quad \forall k \in S_s\\
&\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}v) \mathrel{\widehat{=}} \mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}v, \quad \forall v \in V_s\\
&\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}(\alpha,\beta)) \mathrel{\widehat{=}}\\
&\qquad (\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\alpha), \text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\beta))\\
&\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\square\alpha{:}\mathcal{A}) \mathrel{\widehat{=}}\\
&\qquad \square\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\alpha){:}\,\langle\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\mathcal{A}_i)\rangle_{i=1}^{\text{size(type}(\alpha))}\\
&\text{pushdown}(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\bot_s) \mathrel{\widehat{=}} \mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\bot_s
\end{aligned}$$

4.9 DEFINITION evaluate$^*_t$: $\mathbf{E}^\star \longrightarrow \mathbf{E}^\star_t$

$t \in \mathbb{N}$

Evaluates an $\mathbf{E}^\star$ expression at time t.

$\forall n \geq 0$

$$\begin{aligned}
&\text{evaluate}^*_t(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}k) \mathrel{\widehat{=}} k, \quad \forall k \in S_s,\ t \geq n\\
&\text{evaluate}^*_t(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}k) \mathrel{\widehat{=}} k_{t+1}, \quad \forall k \in S_s,\ t < n\\
&\text{evaluate}^*_t(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}v) \mathrel{\widehat{=}} v_{t-n}, \quad \forall v \in V_s,\ t \geq n\\
&\text{evaluate}^*_t(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}v) \mathrel{\widehat{=}} k_{t+1}, \quad \forall v \in V_s,\ t < n\\
&\text{evaluate}^*_t(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\bot_s) \mathrel{\widehat{=}} \bot_s,\ t \geq n\\
&\text{evaluate}^*_t(\mathbf{\Delta}_{k_1}\ldots\mathbf{\Delta}_{k_n}\bot_s) \mathrel{\widehat{=}} k_t,\ t < n\\
&\text{evaluate}^*_t((\alpha,\beta)) \mathrel{\widehat{=}} (\text{evaluate}^*_t(\alpha), \text{evaluate}^*_t(\beta))\\
&\text{evaluate}^*_t(\square\alpha{:}\mathcal{A}) \mathrel{\widehat{=}} \square\text{evaluate}^*_t(\alpha){:}\,\left\langle\text{evaluate}^*_t(\mathcal{A}_1)\ldots\text{evaluate}^*_t(\mathcal{A}_{\text{size(type}(\alpha))})\right\rangle
\end{aligned}$$

$\mathbf{E}^\star_t$ represents $\mathbf{E}^\star$ restricted to time t. Expressions in $\mathbf{E}^\star_t$ no longer contain delays, and all variables in such expressions have an index which gives the time ($\leq t$) at which their value is to be taken. Note however that variables are retained: they are not given a constant valuation. So an expression in $\mathbf{E}^\star$ of depth + period d can be represented by its evaluations in $\mathbf{E}^\star_0$ to $\mathbf{E}^\star_d$.

Note how 'evaluate$^*_t$' operates only on expressions in the form produced by 'pushdown', ie with delays at the roots of the expression.

4.10 DEFINITION $\quad \text{formperm} : \mathbf{E}_t^\star \longrightarrow \mathbf{E}_t^\star$

Transforms □'s into constant permutations.

$$\begin{array}{l}
\text{formperm}(k) \mathrel{\widehat{=}} l, \quad \forall k \in S_s \\
\text{formperm}(v_t) \mathrel{\widehat{=}} v_t, \quad \forall v \in V_s \\
\text{formperm}((\alpha, \beta)) \mathrel{\widehat{=}} (\text{formperm}(\alpha), \text{formperm}(\beta)) \\
\text{formperm}(\Box\alpha{:}\langle \mathcal{A} \ldots \mathcal{A}\rangle) \mathrel{\widehat{=}} \mathcal{A} \\
\text{formperm}(\Box\alpha{:}\langle K_1, \ldots, K_{i-1}, \mathcal{A}_i, \ldots, \mathcal{A}_n\rangle) \mathrel{\widehat{=}} \\
\qquad\qquad \text{formperm}\left(\Box(\mathcal{A}_i, \alpha){:}\langle K_1\rangle_1^m \sqcup \ldots \sqcup \langle K_{i-1}\rangle_1^m \sqcup \overline{S}_{\text{type}(\mathcal{A}_i)} \sqcup \ldots \sqcup \langle \mathcal{A}_n\rangle_1^m\right) \\
\qquad \text{where } m = \text{size}(\text{type}(\mathcal{A}_i)) \\
\qquad\qquad K_j \in \times \bigcup_{s \in \mathbf{S}}(\bot_s \cup S_s) \\
\text{formperm}(\Box\alpha{:}\langle K_1 \ldots K_n\rangle) \mathrel{\widehat{=}} \Box\text{formperm}(\alpha){:}\langle k_1 \ldots k_n\rangle \\
\text{formperm}(\bot_s) \mathrel{\widehat{=}} \bot_s
\end{array}$$

A *permutation* is a case expression with all variables moved into the chooser; eg

$$\begin{array}{l}
\Box(\mathsf{x}, (\mathsf{y}, \mathsf{z})){:} \\
\quad \langle(\mathsf{T}, \mathsf{T}), (\mathsf{T}, \mathsf{T}), (\mathsf{F}, \bot_{\mathsf{BOOL}}), (\bot_{\mathsf{BOOL}}, \bot_{\mathsf{BOOL}}), (\mathsf{F}, \mathsf{F}), (\mathsf{T}, \mathsf{F}), (\mathsf{F}, \mathsf{F}), (\mathsf{F}, \mathsf{F})\rangle
\end{array}$$

I call case expressions in such a form *permutations* since they permute their chooser: for example, if n belongs to the enumerated type $\langle 1, 2, 3\rangle$ then $\Box\mathsf{n}{:}\langle 2, 3, 1\rangle$ has a similar meaning to the mathematical permutation $(1, 2, 3)n$. Notice that in line 4 of the above definition, the identity permutation is removed.

4.11 DEFINITION $\quad \text{canon} : \mathbf{E}_t^\star \longrightarrow \mathbf{E}_t^\star$

Reduces $\mathbf{E}_t^\star$ *to a canonical form.*

$$\begin{array}{l}
\text{canon}(k) \mathrel{\widehat{=}} k, \quad \forall k \in S_s \\
\text{canon}(v_t) \mathrel{\widehat{=}} v_t, \quad \forall v \in V_s \\
\text{canon}(\bot_s) \mathrel{\widehat{=}} \bot_s \\
\text{canon}((\Box\alpha{:}\mathcal{A}, k)) \mathrel{\widehat{=}} \Box\alpha{:}\langle(\mathcal{A}_1, k) \ldots (\mathcal{A}_n, k)\rangle \\
\text{canon}((\Box\alpha{:}\mathcal{A}, v_t)) \mathrel{\widehat{=}} \\
\qquad \text{contract}\left(\text{canon}\left(\Box v_t{:}\left\langle \Box\alpha{:}\langle(\mathcal{A}_1, (\overline{S}_{\text{type}(v)})_i) \ldots (\mathcal{A}_n, (\overline{S}_{\text{type}(v)})_i)\rangle\right\rangle_{i=1}^{\text{size}(\text{type}(v))}\right)\right) \\
\text{canon}((\Box\alpha{:}\mathcal{A}, \bot_s)) \mathrel{\widehat{=}} \Box\alpha{:}\langle(\mathcal{A}_1, \bot_s) \ldots (\mathcal{A}_n, \bot_s)\rangle \\
\text{canon}((\Box\alpha{:}\mathcal{A}, (\beta, \gamma))) \mathrel{\widehat{=}} \\
\qquad \text{canon}\left(\Box(\beta, \gamma){:}\left\langle \Box\alpha{:}\langle(\mathcal{A}_1, (\overline{S}_{\text{type}((\beta,\gamma))})_i) \ldots (\mathcal{A}_n, (\overline{S}_{\text{type}((\beta,\gamma))})_i)\rangle\right\rangle_{i=1}^{\text{size}(\text{type}((\beta,\gamma)))}\right) \\
\text{canon}((\Box\alpha{:}\mathcal{A}, \Box\beta{:}\mathcal{B})) \mathrel{\widehat{=}} \text{canon}\left(\Box\alpha{:}\langle \Box\beta{:}\langle(\mathcal{A}_i, \mathcal{B}_1) \ldots (\mathcal{A}_i, \mathcal{B}_n)\rangle\rangle_{i=1}^{\text{size}(\text{type}(\alpha))}\right) \\
\text{canon}((k, \Box\alpha{:}\mathcal{A})) \mathrel{\widehat{=}} \Box\alpha{:}\langle(k, \mathcal{A}_1) \ldots (k, \mathcal{A}_n)\rangle \\
\text{canon}((v_t, \Box\alpha{:}\mathcal{A})) \mathrel{\widehat{=}} \\
\qquad \text{contract}\left(\text{canon}\left(\Box v_t{:}\left\langle \Box\alpha{:}\langle((\overline{S}_{\text{type}(v)})_i, \mathcal{A}_1) \ldots ((\overline{S}_{\text{type}(v)})_i, \mathcal{A}_n)\rangle\right\rangle_{i=1}^{\text{size}(\text{type}(v))}\right)\right) \\
\text{canon}((\bot_s, \Box\alpha{:}\mathcal{A})) \mathrel{\widehat{=}} \Box\alpha{:}\langle(\bot_s, \mathcal{A}_1) \ldots (\bot_s, \mathcal{A}_n)\rangle
\end{array}$$

$$
\begin{aligned}
&\mathrm{canon}(((\alpha,\beta),\Box\gamma{:}\mathcal{A})) \ \widehat{=}\\
&\qquad \mathrm{canon}\Big(\Box(\alpha,\beta){:}\ \Big\langle\Box\gamma{:}\langle((\overline{S}_{\mathrm{type}((\alpha,\beta))})_i,\mathcal{A}_1)\ldots((\overline{S}_{\mathrm{type}((\alpha,\beta))})_i,\mathcal{A}_n)\rangle\Big\rangle_{i=1}^{\mathrm{size}(\mathrm{type}((\alpha,\beta)))}\Big)\\
&\mathrm{canon}((\alpha,\beta)) \ \widehat{=}\ (\alpha,\beta)\\
&\mathrm{canon}(\Box\alpha{:}\langle\mathcal{A}\ldots\mathcal{A}\rangle) \ \widehat{=}\ \mathcal{A}\\
&\mathrm{canon}(\Box k{:}\mathcal{A}) \ \widehat{=}\ \mathrm{canon}(\mathcal{A}_{\mathrm{n}(k)})\\
&\mathrm{canon}(\Box v_t{:}\mathcal{A}) \ \widehat{=}\ \Box v_t{:}\ \Big\langle\mathrm{canon}(\mathcal{A}_1[v_t\backslash(\overline{S}_{\mathrm{type}(v)})_1])\ldots\mathrm{canon}(\mathcal{A}_n[v_t\backslash(\overline{S}_{\mathrm{type}(v)})_n])\Big\rangle\\
&\mathrm{canon}(\Box\bot_s{:}\mathcal{A}) \ \widehat{=}\ \bot_{\mathrm{type}(\mathcal{A}_1)}\\
&\mathrm{canon}(\Box(\alpha,\beta){:}\mathcal{A}) \ \widehat{=}\ \mathrm{contract}\Big(\mathrm{canon}(\Box\beta{:}\ \Big\langle\Box\alpha{:}\langle\mathcal{A}_{ni+1}\ldots\mathcal{A}_{n(i+1)}\rangle\Big\rangle_{i=0}^{m-1})\Big)\\
&\qquad\qquad \text{where } n=\mathrm{size}(\mathrm{type}(\alpha)),\ m=\mathrm{size}(\mathrm{type}(\beta))\\
&\mathrm{canon}(\Box\Box\alpha{:}\mathcal{A}{:}\mathcal{B}) \ \widehat{=}\ \Box\alpha{:}\langle\mathrm{canon}(\Box\mathcal{B}_1{:}\mathcal{A})\ldots\mathrm{canon}(\Box\mathcal{B}_n{:}\mathcal{A})\rangle
\end{aligned}
$$

'Canon' operates on $\mathbf{E}_t^\star$ expressions which have been processed by 'formperm'. It reduces expressions to a canonical form. The canonical form is that of a single large permutation, with all input variables paired together in alphabetical order in the chooser. The chooser is also flattened so that pairs can only occur as the right hand component of other pairs:

$$\Box(\mathrm{a}_t,(\mathrm{b}_t,(\mathrm{c}_t,(\ldots)))){:}\langle K_1,\ldots,K_n\rangle$$

All combinatorial circuits, and therefore all expressions in $\mathbf{E}_t^\star$, can be represented uniquely in this form. The function 'canon' uses a number of standard though lengthy transformations on $\mathbf{E}_t^\star$ to attain this form.

4.12 DEFINITION contract : $\mathbf{E}_t^\star \longrightarrow \mathbf{E}_t^\star$

Contracts nested □'s into a single □.

$$
\begin{aligned}
&\mathrm{contract}(\Box v_t{:}\langle\Box w_t{:}\mathcal{A}_i\rangle) \ \widehat{=}\\
&\qquad \mathrm{contract}\,(\Box(w_t,v_t){:}\mathcal{A}_1\sqcup\ldots\sqcup\mathcal{A}_n) && w<v\\
&\qquad \mathrm{contract}\,\Big(\Box(v_t,w_t){:}\langle(\mathcal{A}_1)_i\ldots(\mathcal{A}_n)_i\rangle_{i=1}^{\mathrm{size}(\mathrm{type}(w))}\Big) && w>v\\
&\mathrm{contract}(\Box(v_t,\alpha){:}\langle\Box w_t{:}\mathcal{A}_i\rangle) \ \widehat{=}\\
&\qquad \mathrm{contract}\,(\Box(w_t,(v_t,\alpha)){:}\mathcal{A}_1\sqcup\ldots\sqcup\mathcal{A}_n) && w<v\\
&\qquad \mathrm{contract}\Big(\Box v_t{:}\ \Big\langle\mathrm{contract}(\Box\alpha{:}\langle\Box w_t{:}\mathcal{A}_{ij+1}\rangle_{j=0}^{\mathrm{size}(\mathrm{type}(\alpha))-1})\Big\rangle_{i=1}^{\mathrm{size}(\mathrm{type}(v))}\Big) && w>v\\
&\mathrm{contract}(\Box\alpha{:}\langle K_1\ldots K_n\rangle) \ \widehat{=}\ \Box\alpha{:}\langle K_1\ldots K_n\rangle\\
&\qquad\qquad \forall K_i\in\times^{\bigcup_{s\in\mathbf{S}}(\bot_s\cup S_s)}\\
&\mathrm{contract}(\Box v_t{:}\langle\Box(\alpha,\beta){:}A_i\rangle) \ \widehat{=}\ \Box(v_t,(\alpha,\beta)){:}\langle(\mathcal{A}_1)_i\ldots(\mathcal{A}_n)_i\rangle_{i=1}^{\mathrm{size}(\mathrm{type}((\alpha,\beta)))}
\end{aligned}
$$

'Canon' expands cases where the chooser is a pair into a case of a single variable where the expressions in the choice sequence are 'internal' cases of similar form. It does this so that it can easily determine redundancy. For example, the expression:

$$\Box(\mathrm{v},\mathrm{v}){:}\langle K_1,\ldots,K_4\rangle$$

is expanded to:

$$\square \mathsf{v}{:}\,\langle \square \mathsf{v}{:}\langle K_1, K_2\rangle, \square \mathsf{v}{:}\langle K_3, K_4\rangle\rangle$$

'Canon' is then called again on the result; inside the outer case operator it subtitutes all occurrences of the outer chooser v by constants:

$$\square \mathsf{v}{:}\,\langle \square \mathsf{T}{:}\langle K_1, K_2\rangle, \square \mathsf{F}{:}\langle K_3, K_4\rangle\rangle$$

Next the function can simplify the constant-chooser case expressions:

$$\square \mathsf{v}{:}\langle K_1, K_4\rangle$$

The final expression is in the required form. However, in general not all variables will be eliminated because they are redundant, as in this case. If the original equation had instead been:

$$\square(\mathsf{w}, \mathsf{v}){:}\langle K_1, ..., K_4\rangle$$

Then 'canon' would get as far as producing

$$\square \mathsf{v}{:}\,\langle \square \mathsf{w}{:}\langle K_1, K_2\rangle, \square \mathsf{w}{:}\langle K_3, K_4\rangle\rangle$$

But could obviously simplify this no further. This expression is not however in the required canonical form. The function 'contract' is required to bring the inner case expressions out once more:

$$\square(\mathsf{v}, \mathsf{w}){:}\langle K_1, K_3, K_2, K_4\rangle$$

Note that 'contract' brings w back into the main chooser in alphabetical order, so that in this instance the order of the sequence of constants has changed so as to preserve the sense.

4.13 DEFINITION $\mathrm{parse}_f : \mathbf{E}_t^\star \longrightarrow \mathbf{E}_t^\star$

$f : \mathbf{E}_t^\star \longrightarrow \mathbf{E}_t^\star$

Applies f to the parse-tree of $\mathbf{E}_t^\star$ expressions

$$\begin{aligned}
\mathrm{parse}_f(k) &\mathrel{\hat{=}} f(k), \quad \forall k \in S_s\\
\mathrm{parse}_f(v_t) &\mathrel{\hat{=}} f(v_t), \quad \forall v \in V_s\\
\mathrm{parse}_f((\alpha, \beta)) &\mathrel{\hat{=}} f((\mathrm{parse}_f(\alpha), \mathrm{parse}_f(\beta)))\\
\mathrm{parse}_f(\square\alpha{:}\mathcal{A}) &\mathrel{\hat{=}} f(\square\mathrm{parse}_f(\alpha){:}\mathcal{A})\\
\mathrm{parse}_f(\bot_s) &\mathrel{\hat{=}} f(\bot_s)
\end{aligned}$$

'Parse' is needed because the function 'canon' expects that, when it is applied to an expression, the components of that expression are already in canonical form. That is, it must be applied from the roots of an expression up. 'parse_{canon}' is the function which applies 'canon' in this way.

The main reason why this is necessary is that 'canon' must float case operators to the outside of pairs. If the function is given a pair, it must know that there are no case expressions hidden deep within the pair's structure. Hence it must be applied from the roots of the pair upward to float any cases up through the structure.

4.14 DEFINITION $\equiv_d^*$

$d \in \mathbf{N}$

Gives equivalence between two expression in $\mathbf{E}^*$ *to depth d.*

$$\alpha \equiv_d^* \beta \triangleq \mathrm{parse}_{\mathrm{canon}}(\mathrm{evaluate}_t^*(\mathrm{pushdown}(\alpha))) = \mathrm{parse}_{\mathrm{canon}}(\mathrm{evaluate}_t^*(\mathrm{pushdown}(\beta)))$$

$$\forall t < d$$

5. The Equivalence Theorem

5.1 THEOREM $\alpha \equiv \beta$ *iff* $\mathrm{reduce}_d(\alpha) \equiv_d^* \mathrm{reduce}_d(\beta)$
where $d \triangleq \mathrm{depth}(\alpha)+\mathrm{depth}(\beta)+\max\left\{\mathrm{period}_{\emptyset,0}(\alpha),\ \mathrm{period}_{\emptyset,0}(\beta)\right\}$

Proof:

There are two stages to the proof – the first is to show that the depth d of expansion defined above is sufficient to preserve uniqueness of expressions; the second is to show that equivalence in $\mathbf{E}^*$ as defined in **4.14** corresponds to equivalence in **E** for expanded expressions. This is just a matter of showing that the functions 'pushdown', 'formperm' and 'canon' do not alter the meaning of expressions, and that the final canonical form is indeed a unique representation of expressions. The latter is clear from the nature of the form, and from the fact that, if the canonization functions are indeed correct, then they are certainly able to transform any expression into the form.

To show the *canonization functions* are correct, we must prove the validity of every line of these functions. For example, to show that the line

$$\mathrm{canon}((\Box\alpha{:}\mathcal{A}, k)) \ \triangleq\ \Box\alpha{:}\langle(\mathcal{A}_1, k)\ldots(\mathcal{A}_n, k)\rangle$$

is correct, we must demonstrate that, <u>in **E**</u>,

$$(\Box\alpha{:}\mathcal{A}, k) \equiv \Box\alpha{:}\langle(\mathcal{A}_1, k)\ldots(\mathcal{A}_n, k)\rangle$$

This is done by using '$\mathrm{evaluate}_{t,V}$' on each side of the equivalence:

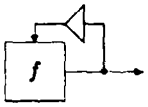
$$\begin{aligned}\mathrm{evaluate}_{t,V}(\Box\alpha{:}\langle(\mathcal{A}_1, k)\ldots(\mathcal{A}_n, k)\rangle) &= \mathrm{evaluate}_{t,V}((\mathcal{A}_i, k))\\ &= (\mathrm{evaluate}_{t,V}(\mathcal{A}_i), \mathrm{evaluate}_{t,V}(k))\\ &= (\mathrm{evaluate}_{t,V}(\Box\alpha{:}\mathcal{A}), \mathrm{evaluate}_{t,V}(k))\\ &= \mathrm{evaluate}_{t,V}((\Box\alpha{:}\mathcal{A}, k))\end{aligned}$$

For some some i dependant on V; for all V. All lines of the functions 'pushdown', 'formperm', 'canon' and 'contract' can be proved in this way.

The difficult part of the proof is showing the depth of expansion is sufficient. We must show that no expression in **E** can produce a sequence of outputs whose period is greater than that predicted by the 'period' function. An idea of the proof of the validity of this function for case and pair expressions has already been given, in section **3.5**. But the formal derivation of the period of a recursive expression ought to be given.

Consider a single loop represented by

$$v_t = f(v_{t-1})$$



The loop function f depends on only one variable, so that it can do at most p different things, where p is the size of the type of v. Clearly the period of this function is therefore at most p. In general a single loop can be represented by

$$v_t = f(v_{t-1}, v_{t-2}, \ldots, v_{t-n}).$$

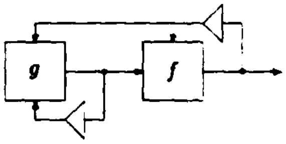
Here the function f depends on n variables, so that its period is at most p^n.

Now for the case of two loops, one inside the other. We have, for example,

$$v_t = f(v_{t-1}, w_t)$$

where

$$w_t = g(w_{t-1}, v_{t-1}).$$



Substituting,

$$v_t = f(v_{t-1}, g(w_{t-1}, v_{t-1})).$$

This can be rewritten as a single function, say h:

$$v_t = h(v_{t-1}, w_{t-1}).$$

Clearly the maximum period of this is pq where p is again the size of the type of v, and q the size of the type of w. This is indeed the result that 'period' would produce.

This argument can easily be generalized to deal with any number of nested loops with any number of delays in their feedback.

6. Conclusion

Although lengthy, I am reasonably confident of the accuracy of the equivalence testing process described above. However there are a couple of problems in the application of this theory which are still to be overcome.

Complexity

The main cause for concern is in the expansion of **E** expressions. For non-trivial circuits with complicated nesting, the expanded expression becomes extremely large very quickly. However, this growth may be limited in a clever implementation of the expander, by considering that the roots of the expanded expression are all the same. Hence tree-storage optimization is applicable to reduce the size of the expanded expression to near the size of the original.

The ideal form in which to implement all the functions described above would of course be a functional language. A good functional language (see [4]) only requires storage for the root of the expression it is currently processing. This again avoids the explosion of storage of the expanded expression.

Data Refinement

The theory described can only compare expressions with the same input variables. If it is desired to test equivalence between one circuit and another which is thought to be a data-refinement of the first, then a function must be added around the latter which converts the original input variables to the new form. For example, it might be necessary to convert separate read and write lines into a single read/write signal by use of a case expression.

The Undefined Value

In the equivalence theory, bottoms or tops must be inaccessible to the user; otherwise some of the transformations in functions such as 'canon' are invalid. Any undefined values needed must therefore be just further type constants and included in their S_s. This may have consequences for how such values are considered by Ella at the moment.

This concludes the description of the equivalence theory. I would like to acknowledge the contributions of John Morison, who gave me the idea for the research in the first place; and Roy Milner and Ian Currie for their constructive ideas and criticisms during the progress of the work.

Mark Davies, 1988.

A. Technical Definitions

A.15 DEFINITION $\mathrm{type} : \mathbf{E} \longrightarrow \times^{\mathbf{S}}$

Returns the type of its argument.

$$\begin{aligned}
\mathrm{type}(k) &\;\widehat{=}\; s, \quad \forall k \in S_s \\
\mathrm{type}(v) &\;\widehat{=}\; s, \quad \forall v \in V_s \\
\mathrm{type}(\Delta_k \alpha) &\;\widehat{=}\; \mathrm{type}(k) \\
\mathrm{type}((\alpha, \beta)) &\;\widehat{=}\; (\mathrm{type}(\alpha), \mathrm{type}(\beta)) \\
\mathrm{type}(\square \alpha{:}\mathcal{A}) &\;\widehat{=}\; \mathrm{type}(\mathcal{A}_1) \\
\mathrm{type}(\mu v.\alpha) &\;\widehat{=}\; \mathrm{type}(v)
\end{aligned}$$

A.16 DEFINITION $\mathrm{size} : \times^{\mathbf{S}} \longrightarrow \mathbf{N}$

Returns the cross-product size of a type.

$$\begin{aligned}
\mathrm{size}(s) &\;\widehat{=}\; \#S_s, \quad \forall s \in \mathbf{S} \\
\mathrm{size}(s, t) &\;\widehat{=}\; \mathrm{size}(s) \times \mathrm{size}(t)
\end{aligned}$$

A.17 DEFINITION $\mathrm{welldef}_{\square} : \mathbf{E} \times 2^{\mathbf{E}} \longrightarrow \mathbf{B}$

Ensures correct number and type of expressions in □ output.

$$\begin{aligned}
\mathrm{welldef}_{\square}(\alpha, \mathcal{A}) \;\widehat{=}\; & \mathcal{A} = \langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n \rangle \\
& n = \mathrm{size}(\mathrm{type}(\alpha)) \\
& \mathrm{type}(\mathcal{A}_i) = \mathrm{type}(\mathcal{A}_j) \;\; \forall i, j = 1 \ldots n
\end{aligned}$$

A.18 DEFINITION $\mathrm{welldef}_{\mu} : V_s \times \mathbf{E} \longrightarrow \mathbf{B}$

Ensures recursive expression is well typed and will not spin.

$$\begin{aligned}
\mathrm{welldef}_{\mu}(v, \alpha) \;\widehat{=}\; & \exists s \in \mathbf{S} \;\; \text{s.t. } \mathrm{type}(\alpha) = s = \mathrm{type}(v) \\
& \mathrm{no}_v(\alpha)
\end{aligned}$$

A.19 DEFINITION $\mathbf{no}_v : \mathbf{E} \longrightarrow \mathbf{B}$

$v \in V_s$

Checks there are no undelayed v variables in expression.

$$\begin{array}{l}
\mathbf{no}_v(k) \ \widehat{=}\ \mathbf{True}, \quad \forall k \in S_s \\
\mathbf{no}_v(w) \ \widehat{=}\ \mathbf{True}, \quad \forall w \in V_s,\ w \neq v \\
\mathbf{no}_v(v) \ \widehat{=}\ \mathbf{False} \\
\mathbf{no}_v(\boldsymbol{\Delta}_k\alpha) \ \widehat{=}\ \mathbf{nodum}_v(\alpha) \\
\mathbf{no}_v((\alpha,\beta)) \ \widehat{=}\ \mathbf{no}_v(\alpha) \wedge \mathbf{no}_v(\beta) \\
\mathbf{no}_v(\Box\alpha{:}\mathcal{A}) \ \widehat{=}\ \mathbf{no}_v(\alpha) \wedge \bigwedge_{i=1}^{\mathbf{size(type(\alpha))}} \mathbf{no}_v(\mathcal{A}_i) \\
\mathbf{no}_v(\mu w.\alpha) \ \widehat{=}\ \mathbf{no}_v(\alpha),\ w \neq v \\
\mathbf{no}_v(\mu v.\alpha) \ \widehat{=}\ \mathbf{False}
\end{array}$$

A.20 DEFINITION $\mathbf{nodum}_v : \mathbf{E} \longrightarrow \mathbf{B}$

$v \in V_s$

Checks there are no v dummy recursion variables.

$$\begin{array}{l}
\mathbf{nodum}_v(k) \ \widehat{=}\ \mathbf{True}, \quad \forall k \in S_s \\
\mathbf{nodum}_v(w) \ \widehat{=}\ \mathbf{True}, \quad \forall w \in V_s \\
\mathbf{nodum}_v(\boldsymbol{\Delta}_k\alpha) \ \widehat{=}\ \mathbf{nodum}_v(\alpha) \\
\mathbf{nodum}_v((\alpha,\beta)) \ \widehat{=}\ \mathbf{nodum}_v(\alpha) \wedge \mathbf{nodum}_v(\beta) \\
\mathbf{nodum}_v(\Box\alpha{:}\mathcal{A}) \ \widehat{=}\ \mathbf{nodum}_v(\alpha) \wedge \bigwedge_{i=1}^{\mathbf{size(type(\alpha))}} \mathbf{nodum}_v(\mathcal{A}_i) \\
\mathbf{nodum}_v(\mu w.\alpha) \ \widehat{=}\ \mathbf{nodum}_v(\alpha),\ w \neq v \\
\mathbf{nodum}_v(\mu v.\alpha) \ \widehat{=}\ \mathbf{False}
\end{array}$$

A.21 DEFINITION $\times^{\mathbf{S}}$ **is the least set s.t.**

$$\begin{array}{l}
\mathbf{S} \subset \times^{\mathbf{S}} \\
(s,t) \in \times^{\mathbf{S}}, \quad \forall s,t \in \times^{\mathbf{S}}
\end{array}$$

A.22 DEFINITION $\overline{S}_T$ is defined for $T \in \times^{\mathbf{S}}$ as follows:

$$\begin{array}{l}
\overline{S}_s \ \widehat{=}\ S_s,\ \forall s \in \mathbf{S} \\
\overline{S}_{(T,T')} \ \widehat{=}\ \bigsqcup_{i=1}^{\#\overline{S}_{T'}} \left\langle ((\overline{S}_T)_1, (\overline{S}_{T'})_i) \ldots ((\overline{S}_T)_n, (\overline{S}_{T'})_i) \right\rangle \\
\qquad \forall T, T' \in \times^{\mathbf{S}}
\end{array}$$

A.23 DEFINITION $\mathbf{E}^{\star}$ is the least set s.t.

$$\begin{array}{ll}
S_s \subset \mathbf{E}^{\star} & \forall s \in \mathbf{S} \\
V_s \subset \mathbf{E}^{\star} & \forall s \in \mathbf{S} \\
\boldsymbol{\Delta}_k\alpha \in \mathbf{E}^{\star} & \forall \alpha \in \mathbf{E}^{\star}, \forall k \in S_s \text{ s.t. } \mathrm{type}(\alpha) = s \\
(\alpha,\beta) \in \mathbf{E}^{\star} & \forall \alpha,\beta \in \mathbf{E}^{\star} \\
\Box\alpha{:}\mathcal{A} \in \mathbf{E}^{\star} & \forall \alpha \in \mathbf{E}^{\star}, \forall \mathcal{A} \subset \mathbf{E}^{\star} \text{ s.t. } \mathrm{welldef}_{\Box}(\alpha,\mathcal{A})) \\
\bot_s \in \mathbf{E}^{\star} & \forall s \in \mathbf{S}
\end{array}$$

B. References

[1] "Sequential Programming Extensions to Ella, with Automatic Transformation to Structure": J.D.Morison, N.E.Peeling, E.V.Whiting.

[2] "Transformations of Ella": E.V.Whiting, D.C.Taylour (in preparation).

[3] Ella Language Reference Manual: Praxis Systems PLC, Bath.

[4] Orwell Manual: Programming Research Group, Oxford.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ...UNCLASSIFIED... ....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 4225	3. Agency Reference	4. Report Security Classification Unclassified

5. Originator's Code (if known) 7784000	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment St Andrews Road, Malvern, Worcestershire WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

MATHEMATICAL EQUIVALENCE IN A PRIMITIVE ELLA

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Davies M B	9(a) Author 2	9(b) Authors 3,4...	10. Date 1988.8	pc. ref. 22

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The mathematical language L is defined, which represents circuits in a primitive subset of Ella containing delays, pairing, CASE (multiplexer) expressions, and recursive or feedback expressions. It is shown how by reduction of expressions in this language to an approximated finite form, equivalence between expressions can be tested in finite time.